Modular Verification of Deadlock Freedom in the Presence of Condition Variables

Jafar Hamin *Bart Jacobs*

Report CW 704, May 2017

KU Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Modular Verification of Deadlock Freedom in the Presence of Condition Variables

Jafar Hamin *Bart Jacobs*

Report CW 704, May 2017

Department of Computer Science, KU Leuven

Abstract

One of the common methods to synchronize threads in multi-threaded programs is using condition variables, where calling the wait command for a condition variable blocks the caller thread and signaling a condition variable wakes all the blocked threads waiting for that condition variable up. One potential problem with these programs is that a waiting thread might be blocked forever leading the program to deadlock. In this report a verification approach for deadlock freedom of such programs is presented. In this approach each condition variable has a corresponding *ghost* value, namely *cvg*, such that when a condition variable is created and a wait command for that variable is called the corresponding *cvg* is non-negative and non-positive, respectively. Calling a wait command for a condition variable also requires a credit for that variable, where a credit for a condition variable can be achieved when a thread increments the corresponding *cvg* and also signals that condition variable if the old value of *cvg* was zero. We are now developing a Coq formalization to formally prove soundness of the presented approach

Modular Verification of Deadlock Freedom in the Presence of Condition Variables*

Jafar Hamin
imec-DistriNet, Dept. C.S., KU Leuven
Celestijnenlaan 200A, 3001 Leuven, Belgium
jafar.hamin@cs.kuleuven.be

Bart Jacobs
imec-DistriNet, Dept. C.S., KU Leuven
Celestijnenlaan 200A, 3001 Leuven, Belgium
bart.jacobs@cs.kuleuven.be

Abstract

One of the common methods to synchronize threads in multi-threaded programs is using condition variables, where calling the wait command for a condition variable blocks the caller thread and signaling a condition variable wakes all the blocked threads waiting for that condition variable up. One potential problem with these programs is that a waiting thread might be blocked forever leading the program to deadlock. In this report a verification approach for deadlock freedom of such programs is presented. In this approach each condition variable has a corresponding *ghost* value, namely *cvg*, such that when a condition variable is created and a wait command for that variable is called the corresponding *cvg* is non-negative and non-positive, respectively. Calling a wait command for a condition variable also requires a credit for that variable, where a credit for a condition variable can be achieved when a thread increments the corresponding *cvg* and also signals that condition variable if the old value of *cvg* was zero. We are now developing a Coq formalization to formally prove soundness of the presented approach.

1 Introduction

Recently, there has been growing interest in termination verification of programs [1, 2]. Some other verification approaches have been also introduced to prove deadlock freedom and finite blocking of non-terminating programs [3, 4]. To the best of our knowledge, none of the presented approaches are applicable for

*This work was funded by Flemish Research Fund (FWO) grant G.0058.13 and by KU Leuven Research Fund (Onderzoeksraad) grant OT/13/065.

the programs that use condition variables. These variables are widely used to synchronize the execution of threads in multi-threaded environments. However, one potential problem with these programs is infinite blocking of the threads waiting to be signaled that leads to deadlock.

To clarify the problem consider the following *producer/consumer* example in which the condition variable v protects the consumer from reading from an empty buffer. The consumer first acquires the related lock and while there is no item in the buffer it releases the lock, blocks itself and waits for a signal v from the producer. If the consumer wakes up while the buffer is not empty it takes one item out of the buffer and finally releases the related lock. The producer also acquires the same lock, signals all the threads waiting for v if the buffer is currently empty, puts an item on the buffer, and lastly releases the acquired lock.

```
routine main()
{
  q := newqueue;
  l := newlock;
  v := newcvar;
  b := buffer{q:=q, l:=l, v:=v};
  fork (consumer(b));
  producer(b);
  freecvar(b.v);
  freelock(b.l);
  freequeue(b.q);
}
```

```
routine producer(buffer b)
{
  lock_acquire(b.l);
  if (sizeof(b.q) = 0)
    signalAll(b.v);
  enqueue(b.q, "data");
  lock_release(b.l);
}
routine consumer(buffer b)
{
  lock_acquire(b.l);
  while(sizeof(b.q) = 0)
    wait(b.v, b.l);
  dequeue(b.q);
  lock_release(b.l);
}
```

This program is deadlock-free. However, it is easy to construct some variations of this program that deadlock: if there are more than one consumer in the environment, or if the consumer waits for the condition variable even when the buffer is not empty, or if there is a cycle in the wait-for graph of the program.

2 Verification of Deadlock Freedom

In order to verify deadlock freedom of programs that use condition variables, the main idea presented in this paper is to associate each condition variable v with a *ghost* value, namely cvg, such that the value of this variable when v is created is non-negative and when $\mathsf{wait}(v, l)$ is called is non-positive. For the producer/consumer program, for example, the size of the queue is the cvg of the condition variable v, since it meets both requirements. A $\mathsf{wait}(v, l)$ command is successfully verified if the executing thread possesses a *credit* for v, where a credit for a condition variable v can be produced when cvg of v is

incremented and a $\mathsf{signalAll}(v)$ is also executed if the *cvg* changes to 1. It is possible to feed multiple wait commands, even in different threads[1], with this credit. Consequently, if there are some threads in the program waiting for a signal, then there is definitely one thread responsible for signaling these threads. Additionally, no thread is allowed to decrement the *cvg* of v except if it spends one $\mathsf{credit}(v)$ for that decrement. This constrain ensures that if after signaling a condition variable a third thread decrements the value of *cvg* to a non-positive value, leading the waiting threads to be blocked again, then there is another thread responsible for signaling the blocked threads.

The only remaining problem is the existence of a cycle in the wait-for graph of the program. To resolve this problem a waiting level is assigned to each condition variable and also each lock, such that a thread t is allowed to acquire a lock z or wait for a condition variable z if the waiting level of z is strictly lower than the waiting levels of all the locks and condition variables that t is expected to release and signal, respectively. The set of these locks and condition variables is called the *obligations* of the thread t and should be empty when t terminates. Accordingly, a lock is added/removed to/from the set of obligations of the current thread when it is acquired/released. A thread t can also produce a credit for a condition variable if it loads that condition variable to the set of its obligations. The produced credit can be used to feed and verify another thread that requires this credit. The thread t can later unloads this condition variable obligation from the set of its obligations if it either produces and spends a credit for this obligation or delegates this obligation to another thread.

In the remainder of this section we first define the syntax of our programming language and its operational semantics. Then, we introduce the notions of *assertions*, *permission heaps*, and *satisfaction*, a relation relating an assertion to a permission heap. Finally, we present the appropriate proof rules to verify programs against accessing dangling pointers, race conditions and also deadlock. At the end of this section we verify the producer/consumer example according to the presented proof rules.

2.1 Syntax of Programs

An expression can be an integer number z, a variable x, or addition of two other expressions. Each closed expression e can be evaluated to an integer, denoted by $[\![e]\!]$, such that $[\![z]\!]=z, [\![x]\!]=0$, and $[\![e_1+e_2]\!]=[\![e_1]\!]+[\![e_2]\!]$. A command can be $\mathsf{val}\ z$, indicating a single value, or $\mathsf{cons}(e)$, that allocates a part of memory, sets $[\![e]\!]$ as its content, and returns the allocated address, or $\mathsf{free}(e)$, that deallocates the memory at the address $[\![e]\!]$, or $\mathsf{lookup}(e)$, that returns the content of the memory at $[\![e]\!]$, or $\mathsf{mutate}(e_1, e_2)$, that sets the content of the memory at $[\![e_1]\!]$ to $[\![e_2]\!]$, or $\mathsf{fork}(c)$, that forks a new thread c, or $\mathsf{let}\ x{:=}c_1\ \mathsf{in}\ c_2$, that first executes c_1 and then substitutes any occurrence of x in c_2 with the resulting value of execution of c_1, or $\mathsf{newlock}$, that creates a new lock, or $\mathsf{freelock}(l)$, that destroys

[1] It is possible to give a credit to a thread and also get it back when the thread terminates. However, in this version of the paper we do not cover the latter case, where the possessions of the forked thread, such as its credits, are given back to the forking thread.

lock l, or $\mathsf{acquire}(l)$, that acquires lock l if it has not been already acquired, or $\mathsf{release}(l)$, that releases lock l, or $\mathsf{newcvar}$, that creates a new condition variable, or $\mathsf{freecvar}(v)$, that destroys the condition variable v, or $\mathsf{wait}(v, l)$, that releases the lock l and blocks the calling thread until it receives a signal v, or $\mathsf{signalAll}(v)$, that signals all the threads waiting for the condition variable v. The command $\mathsf{asleep}(d, l)$ is an implicit command indicating that the current thread has executed the $\mathsf{wait}(v, l)$ command and is not signaled yet. A command can also be a *ghost command* that does not change the heap and is only used for verification purposes. These commands will be explained in the subsequent sections.

$$
\begin{array}{l}
c \in \mathit{Commands},\ e \in \mathit{Expressions},\ x \in \mathit{Vars},\ z \in \mathbb{Z},\ n \in \mathbb{N} \\
e ::= z \mid x \mid e_1{+}e_2 \\
c ::= \mathsf{val}\ z \mid \mathsf{cons}(e) \mid \mathsf{free}(e) \mid \mathsf{lookup}(e) \mid \mathsf{mutate}(e_1, e_2) \\
\qquad \mid \mathsf{fork}(c) \mid \mathsf{let}\ x := c_1\ \mathsf{in}\ c_2 \\
\qquad \mid \mathsf{newlock} \mid \mathsf{acquire}(x) \mid \mathsf{release}(x) \mid \mathsf{freelock}(x) \\
\qquad \mid \mathsf{newcvar} \mid \mathsf{wait}(x, x') \mid \mathsf{signalAll}(x) \mid \mathsf{freecvar}(x) \\
\qquad \mid \mathsf{asleep}(z, z') \mid \mathit{gc} \\
\mathit{gc} ::= \mathsf{g\_loadobs}(z) \mid \mathsf{g\_unloadobs}(z) \mid \mathsf{g\_loadcredit}(z) \mid \mathsf{g\_unloadcredit}(z) \\
\qquad \mid \mathsf{g\_initlock}(z)
\end{array}
$$

2.2 Semantics of Programs

A small step changes the current *configuration* κ to the new one by executing a thread with the identification id in κ. A configuration is a pair of heaps-thread tables, where heaps and thread tables are some partial functions from locations and thread ids to integers and commands, respectively.

$$
\begin{array}{l}
h \in \mathit{Heaps} = \mathit{Locations} \rightharpoonup \mathbb{Z} \\
t \in \mathit{ThreadTables} = \mathit{ThreadIDs} \rightharpoonup \mathit{Commands} \\
\kappa \in \mathit{Configurations} = \mathit{ThreadTables} \times \mathit{Heaps}
\end{array}
$$

The semantics of programs is defined as follows, where $\mathsf{wkup}(t, v)$ changes all the threads waiting for the condition variable v, indicated by $\mathsf{asleep}(v, l)$, to $\mathsf{acquire}(l)$, and *lift* of a value v w.r.t. the default value v', denoted by $\lfloor v \rfloor_{v'}$, is defined such that $\lfloor v \rfloor_{v'} = v$ and $\lfloor \varnothing \rfloor_{v'} = v'$.

$(t[id{:=}\mathsf{val}\ v], h) \rightsquigarrow^{id} (t[id{:=}\varnothing], h)$
$(t[id{:=}\mathsf{cons}(e), h[z{:=}\varnothing]) \rightsquigarrow^{id} (t[id{:=}\mathsf{val}\ z], h[z{:=}[\![e]\!]])$
$(t[id{:=}\mathsf{free}(e)], h) \rightsquigarrow^{id} (t[id{:=}\mathsf{val}\ 0], h[[\![e]\!]{:=}\varnothing])$
$(t[id{:=}\mathsf{lookup}(e)], h) \rightsquigarrow^{id} (t[id{:=}\mathsf{val}\ \lfloor h\ [\![e]\!] \rfloor_0], h)$
$(t[id{:=}\mathsf{mutate}(e_1, e_2)], h) \rightsquigarrow^{id} (t[id{:=}\mathsf{val}\ 0], h[[\![e_1]\!]{:=}[\![e_2]\!]])$
$(t[id{:=}\mathsf{fork}(c), id'{:=}\varnothing], h) \rightsquigarrow^{id} (t[id{:=}\mathsf{val}\ 0, id'{:=}c], h)$
$(t[id{:=}\mathsf{let}\ x{:=}c_1\ \mathsf{in}\ c_2], h) \rightsquigarrow^{id} (t'[id{:=}\mathsf{let}\ x{:=}c_1'\ \mathsf{in}\ c_2], h')$
  if $(t[id{:=}c_1], h) \rightsquigarrow^{id} (t'[id{:=}c_1'], h')$
$(t[id{:=}\mathsf{let}\ x{:=}\mathsf{val}\ v\ \mathsf{in}\ c], h) \rightsquigarrow^{id} (t[id{:=}c[v/x]], h)$
$(t[id{:=}\mathsf{newlock}], h[z{:=}\varnothing]) \rightsquigarrow^{id} (t[id{:=}\mathsf{val}\ z], h[z{:=}1])$
$(t[id{:=}\mathsf{acquire}\ z], h[z{:=}1]) \rightsquigarrow^{id} (t[id{:=}\mathsf{val}\ 0], h[z{:=}0])$
$(t[id{:=}\mathsf{release}\ z], h) \rightsquigarrow^{id} (t[id{:=}\mathsf{val}\ 0], h[z{:=}1])$
$(t[id{:=}\mathsf{freelock}\ z], h) \rightsquigarrow^{id} (t[id{:=}\mathsf{val}\ 0], h[z{:=}\varnothing])$
$(t[id{:=}\mathsf{newcvar}], h[z{:=}\varnothing]) \rightsquigarrow^{id} (t[id{:=}\mathsf{val}\ z], h[z{:=}0])$
$(t[id{:=}\mathsf{wait}(v, l)], h) \rightsquigarrow^{id} (t[id{:=}\mathsf{asleep}(v, l)], h[l{:=}1])$
$(t[id{:=}\mathsf{signalAll}(v)], h) \rightsquigarrow^{id} (\mathsf{wkup}(t, v)[id{:=}\mathsf{val}\ 0], h])$
$(t[id{:=}\mathsf{freecvar}\ z], h) \rightsquigarrow^{id} (t[id{:=}\mathsf{val}\ 0], h[z{:=}\varnothing])$
$(t[id{:=}gc], h) \rightsquigarrow^{id} (t[id{:=}\mathsf{val}\ 0], h)$

2.3 Permission Heaps

A permission heap is a partial function that maps a *location* to a knowledge on that location with a fraction indicating the percentage of possession of that location by the heap. This knowledge indicates that the given location is the address of an integer, a lock, or a condition variable. For the two last cases it also represents the related waiting level of that lock/condition variable. If the location is the address of a lock it also shows the *invariant* of that lock, the list of condition variables and their waiting levels protected by that lock and also their expected *cvg* values when the lock is going to be released, if it has been already acquired. Note that the lock invariant is a function that given the *cvg* values of its condition variables returns the assertion representing the resources that are protected by that lock.

$\pi \in \mathit{Fractions}$ $\quad$ $o \in \mathit{Obligations} = \mathbb{Z} \to \mathbb{N}$
$w \in \mathit{Wlevels} = \mathbb{N}$ $\quad$ $g \in \mathit{Cvg} = \mathbb{Z} \rightharpoonup \mathbb{Z}$
$i \in \mathit{Invariants} = \mathit{Cvg} \to \mathit{Assertions}$ $\quad$ $D \in \mathit{Cvgdom} = \overline{\mathbb{Z} \times \mathbb{N}}$

$$\begin{array}{rcl} p \in \mathit{PHeaps} & = & \mathit{Locations} \rightharpoonup \mathit{Fractions} \times \mathit{Knowledge} \\ k \in \mathit{Knowledge} & = & \mathbb{Z} + (\mathit{Wlevels} \times ((\varnothing \times \mathit{Inv.}) \times \mathit{Cvgdom} \times (\varnothing + \mathit{Cvg})) + \varnothing) \end{array}$$

Two permission heaps can be added, denoted by $\oplus$, if for each location that is allocated in both heaps the addition of fractions does not exceed one and their knowledge does not contradict each other. These principles are formally defined as follows:

$$
\begin{array}{rcll}
p_1 & \oplus & p_2 & = \; p_2 \oplus p_1 = \lambda z.\ (p_1\ z) \oplus' (p_2\ z) \\
 & & & \quad \text{if } p_1 \perp p_2 \\
(\pi, k) & \oplus' & \varnothing & = \; (\pi, k) \\
(\pi_1, z) & \oplus' & (\pi_2, z) & = \; (\pi_1{+}\pi_2, z) \\
(\pi_1, w, i, D, g) & \oplus' & (\pi_2, w_2, i_2, D_2, \varnothing) & = \; (\pi_1{+}\pi_2, w, i, D, g) \\
(\pi_1, w, i, D, \varnothing) & \oplus' & (\pi_2, w_2, i_2, D_2, \varnothing) & = \; (\pi_1{+}\pi_2, w, i, D, \varnothing) \\
(\pi_1, w, \varnothing) & \oplus' & (\pi_2, w_2, \varnothing) & = \; (\pi_1{+}\pi_2, w, \varnothing)
\end{array}
$$

$$
\begin{array}{rcll}
p_1 & \perp & p_2 & \Leftrightarrow \; \forall z.\ p_1\ z{=}(\pi_1, k_1)\ \wedge\ p_2\ z{=}(\pi_2, k_2) \\
 & & & \quad \Rightarrow \pi_1{+}\pi_2 {\leqslant} 1\ \wedge\ k_1 \perp' k_2 \\
z & \perp' & z' & \Leftrightarrow \; z{=}z' \\
(w_1, i_1, D_1, g) & \perp' & (w_2, i_2, D_2, \varnothing) & \Leftrightarrow \; w_1{=}w_2 \wedge i_1{=}i_2 \wedge D_1{=}D_2 \\
(w_1, i_1, D_1, \varnothing) & \perp' & (w_2, i_2, D_2, \varnothing) & \Leftrightarrow \; w_1{=}w_2 \wedge i_1{=}i_2 \wedge D_1{=}D_2 \\
(w_1, \varnothing) & \perp' & (w_2, \varnothing) & \Leftrightarrow \; w_1{=}w_2
\end{array}
$$

2.4 Satisfaction Relation

An assertion can be either emp, indicating an empty permission heap, or $z \overset{\pi}{\mapsto} z$, indicating a permission heap that only possesses π percentage of z and maps it to z', or $\mathsf{ulock}(z, w, D)$, indicating an uninitialized lock at location z with the waiting level w that protects condition variables (and their waiting levels) in D, or lock, indicating a lock, or locked, indicating an acquired lock, or $\mathsf{obs}(o)$, indicating the obligations that the current thread should do, or $\mathsf{credit}(z)$, indicating the credits in possession of the current thread, or $z \prec o$, indicating z is lower that the waiting levels of all obligations in o, or $\exists$, that existentially quantifies over another assertion, or P, indicating that the proposition P holds, or logical conjunction, logical disjunction, or separating conjunction of two other assertions.

$$
\begin{array}{rl}
a \in \mathit{Assertions} ::= & \mathsf{emp} \mid z \overset{\pi}{\mapsto} z \mid \mathsf{ulock}(z, w, D) \mid \mathsf{lock}(\pi, z, w, i, D) \\
 & \mid \mathsf{locked}(\pi, z, w, i, D, g) \mid \mathsf{obs}(o) \mid \mathsf{credit}(z) \mid z \prec o \\
 & \mid \exists (f{:}(v{:}A) \to a) \mid P \mid a \wedge a \mid a \vee a \mid a * a
\end{array}
$$

2.5 Modeling Permission Heaps

Each assertion models a permission heap p, a multiset of credits d, and m that is either $\varnothing$ or a multiset of obligations. This modeling is indicated as follows,

where $\mathbf{0}{=}\lambda\_.\varnothing$, and $0{=}\lambda\_.0$.

$$
\begin{array}{lcl}
m,d,p \Vdash \mathsf{emp} & \Leftrightarrow & m{=}\varnothing \wedge d{=}0 \wedge p{=}\lambda\_.\varnothing \\
m,d,p \Vdash z \overset{\pi}{\mapsto} z' & \Leftrightarrow & m{=}\varnothing \wedge d{=}0 \wedge p{=}\mathbf{0}[z{:=}(\pi, z')] \\
m,d,p \Vdash \mathsf{ulock}(z,w,D) & \Leftrightarrow & m{=}\varnothing \wedge d{=}0 \wedge p{=}\mathbf{0}[z{:=}(1,w,\varnothing,D,\varnothing) \\
 & & ,\overline{v{:=}(\pi,w_v)}], \text{ where } D{=}\overline{(v,w_v)} \\
m,d,p \Vdash \mathsf{lock}(\pi,z,w,i,D) & \Leftrightarrow & m{=}\varnothing \wedge d{=}0 \wedge p{=}\mathbf{0}[z{:=}(\pi,w,i,D,\varnothing) \\
 & & ,\overline{v{:=}(\pi,w_v)}], \text{ where } D{=}\overline{(v,w_v)} \\
m,d,p \Vdash \mathsf{locked}(\pi,z,w,i,D,g) & \Leftrightarrow & m{=}\varnothing \wedge d{=}0 \wedge p{=}\mathbf{0}[z{:=}(\pi,w,i,D,g) \\
 & & ,\overline{v{:=}(\pi,w_v)}], \text{ where } D{=}\overline{(v,w_v)} \\
m,d,p \Vdash \mathsf{obs}(o) & \Leftrightarrow & m = o \wedge d{=}0 \wedge p{=}\lambda\_.\varnothing \\
m,d,p \Vdash \mathsf{credit}(z) & \Leftrightarrow & m{=}\varnothing \wedge d{=}0[z{:=}1] \wedge p{=}\lambda\_.\varnothing \\
m,d,p \Vdash w \prec o & \Leftrightarrow & \forall z{\in}o.\ \exists \pi, w, k.\ p\ z{=}(\pi,w,k) \wedge w{<}w' \\
m,d,p \Vdash \exists(f) & \Leftrightarrow & \exists z.\ m,d,p \Vdash f\ z \\
m,d,p \Vdash P & \Leftrightarrow & P \\
m,d,p \Vdash a \wedge a' & \Leftrightarrow & m,d,p \Vdash a\ \wedge\ m,d,p \Vdash a' \\
m,d,p \Vdash a \vee a' & \Leftrightarrow & m,d,p \Vdash a\ \vee\ m,d,p \Vdash a' \\
m,d,p \Vdash a * a' & \Leftrightarrow & \exists m_1,m_2,d_1,d_2,p_1,p_2.\ m = m_1 \otimes m_2 \wedge \\
 & & d = \lambda z.\ d_1\ z{+}d_2\ z\ \wedge\ p = p_1 \oplus p_2 \wedge \\
 & & m_1,d_1,p_1 \Vdash a\ \wedge\ m_2,d_2,p_2 \Vdash a' \\
 & & \text{where } m{\oplus}\varnothing{=}\varnothing{\oplus}m{=}m \\
a \vDash a' & \Leftrightarrow & \forall m\ d\ p.\ \ m,d,p \Vdash a\ \Rightarrow\ m,d,p \Vdash a'
\end{array}
$$

2.6 Proof Rules

The following proof rules make sure that 1) no dangling pointer will be accessed, and 2) no race condition will occur, and 3) the program never deadlocks. The rules for the commands $\mathsf{cons}, \mathsf{free}, \mathsf{lookup}$, and mutate ensure that no dangling pointer is accessed and a memory cell cannot be accessed/written when it is concurrently being written/accessed by another thread. The rule fork implies that a thread can delegate a part of its permissions and obligations to the forked thread provided that the forked thread discharges all of the delegated obligations. Executing a $\mathsf{newlock}$ command will produce an uninitialized lock assertion with an arbitrary waiting level for that lock. This assertion can be changed to a normal lock assertion later when its invariant, that must not contain any obs or credit assertion, holds with some non-negative values as the *cvg* values of its condition variables. This transition can be done by the ghost command $\mathsf{g\_initlock}$. The rule $\mathsf{acqure}(l)$ makes sure that the waiting level of l is lower that the waiting levels of obligations of the current thread and l is not a dangling pointer. Additionally, it adds l to the list of obligations, implying that this thread should not wait for any resource with a higher/equal waiting level before releasing that lock. This rule also provides the invariant of the lock for the subsequent commands. The rule $\mathsf{release}(l)$ discharges the obligation l and makes sure that the invariant still holds with some appropriate *cvg* values at the end of the critical section. By the appropriate values we mean those that are not changed since the lock was acquired, or those that are changed under some

specific circumstances. These circumstances are indicated in the rules g_dec and g_inc. The former ensures that no thread decrements cvg of a condition variable v except if it spends one $\mathsf{credit}(v)$ for that decrement, and the latter enables a thread to produce a $\mathsf{credit}(v)$ if that thread increments cvg of v by one while the old value of cvg is not zero.

Given an uninitialized lock assertion of l, the rule newcvar adds the created condition variable with an arbitrary waiting level to the list of the condition variables protected by l. The rule $\mathsf{wait}(v, l)$ makes sure that 1) the cvg value of v is non-positive, and 2) the current thread possesses a $\mathsf{credit}(v)$, and 3) the waiting levels of v and l are lower than the waiting levels of all the thread's obligations except for l (note that the wait command releases the lock l and tries to reacquire it after the thread is signaled), and 4) the invariant of lock l holds, and 5) $\mathsf{wait}(v, l)$ is executed under protection of lock l. The rule $\mathsf{signalAll}(v)$ will produce n instances of $\mathsf{credit}(v)$ provided that cvg of v is incremented by n before releasing the lock. The ghost commands $\mathsf{g\_loado}(v)$ and $\mathsf{uloado}(v)$ can be used to load or unload the condition variable v to the list of obligations and credits of the current thread. The proof rules also cover the existential generalization, frame, and consequence rules used in the Hoare logic and separation logic. Note that $g|_D \Leftrightarrow \mathsf{dom}(g)=\{v|(v,w)\in D\}$, $\mathsf{nocrob}(a) \Leftrightarrow \forall m, d, p.\ m, d, p \vDash a \Rightarrow m=0 \wedge o=\varnothing$, $v\in^1 D \Leftrightarrow \exists w.(v,w)\in D$, $D-^1 v$ removes any element (d, w) from D.

$$\{\mathsf{emp}\}\ \mathsf{val}(z)\ \{\lambda r.\ \mathsf{emp} \wedge r=z\} \qquad \{\mathsf{emp}\}\ \mathsf{cons}(e)\ \{\lambda r.\ r\overset{1}{\mapsto}[\![e]\!]\}$$

$$\{[\![e]\!]\overset{1}{\mapsto}z\}\ \mathsf{free}(e)\ \{\lambda\_.\ \mathsf{emp}\} \qquad \{[\![e]\!]\overset{\pi}{\mapsto}z\}\ \mathsf{lookup}(e)\ \{\lambda r.\ [\![e]\!]\overset{\pi}{\mapsto}z \wedge r=z\}$$

$$\{[\![e_1]\!]\overset{1}{\mapsto}z\}\ \mathsf{mutate}(e_1, e_2)\ \{\lambda r.\ [\![e_1]\!]\overset{1}{\mapsto}[\![e_2]\!]\}$$

$$\frac{\{a * \mathsf{obs}(o)\}\ c\ \{\lambda\_.\ \mathsf{obs}\{\}\}}{\{a * \mathsf{obs}(o \uplus o')\}\ \mathsf{fork}(c)\ \{\lambda\_.\ \mathsf{obs}(o')\}}$$

$$\frac{\{a\}\ c_1\ \{a'\} \qquad \forall z.\ \{a'\ z\}\ c_2[z/x]\ \{a''\}}{\{a\}\ \mathsf{let}\ x{=}c_1\ \mathsf{in}\ c_2\ \{a''\}} \qquad \{\mathsf{emp}\}\ \mathsf{newlock}\ \{\lambda r.\ \mathsf{ulock}(r, w, \{\})\}$$

$$\{\mathsf{lock}(1, l, w, i, \{\})\}\ \mathsf{freelock}(l)\ \{\lambda\_.\ \exists g.\ i(g)\}$$

$$\{a * \mathsf{lock}(\pi, l, w, i, D) * \mathsf{obs}(o) \wedge w \prec o\}\ \mathsf{acquire}(l)$$
$$\{\lambda\_.\ a * \mathsf{obs}(o \uplus \{l\}) * \exists g.\ \mathsf{locked}(\pi, l, w, i, D, g) * i(g) \wedge g|_D\}$$

$$\{\mathsf{locked}(\pi, l, w, i, D, g) * i(g) * \mathsf{obs}(o)\}\ \mathsf{release}(l)$$
$$\{\lambda\_.\ \mathsf{lock}(\pi, l, w, i, D) * \mathsf{obs}(o - \{l\})\}$$

$$\{\mathsf{ulock}(l,w,D)\}\ \mathsf{newcvar}()\ \{\lambda r.\ \mathsf{ulock}(l,w,D{\cup}\{(r,w_r)\})\}$$

$$\{\mathsf{lock}(1,l,w,i,D)\wedge d{\in}^1 D\}\ \mathsf{freecvar}(d)\ \{\lambda\_.\ \mathsf{lock}(1,l,w,i,D{-}^1 d)\}$$

$$\begin{array}{c}\{(a*\mathsf{locked}(\pi,l,w_l,i,D,g)\wedge w_l{\prec}o\wedge\exists w_v.\ (v,w_v){\in}D\wedge w_v{\prec}o)*i(g)\\ *\mathsf{credit}(v)*\mathsf{obs}(o{\uplus}\{\!\{l\}\!\})\wedge g(v){\leqslant}0\}\mathsf{wait}(v,l)\\ \{\lambda\_.\ a*\mathsf{credit}(v)*\mathsf{obs}(o{\uplus}\{\!\{l\}\!\})*\exists g'.\ \mathsf{locked}(\pi,l,w_l,i,D,g')*i(g')\wedge g'|_D\}\end{array}$$

$$\vdash\{\mathsf{locked}(\pi,l,w,i,D,g[v{:=}z])\}\ \mathsf{signalAll}(v)\ \{\mathsf{locked}(\pi,l,w,i,D,g[v{:=}z{+}n])*\{*_{i=1}^{n}\ \mathsf{credit}(v)\}\}$$

$$\frac{\{a\}\ c\ \{a'\}}{\{f*a\}\ c\ \{\lambda r.\ f*a'(r)\}}$$

$$\frac{a\vDash a_1\qquad\{a_1\}\ c\ \{a_1'\}\qquad\forall r.\ a_1'(r)\vDash a'(r)}{\{a\}\ c\ \{a'\}}\qquad\frac{\forall z.\ \{f(z)\}\ c\ \{a\}}{\{\exists f\}\ c\ \{a\}}$$

$$\{\mathsf{obs}(o)\}\ \mathsf{g\_loado}(v)\ \{\lambda\_.\ \mathsf{obs}(o{\uplus}\{\!\{v\}\!\})*\mathsf{credit}(v)\}$$

$$\{\mathsf{obs}(o{\uplus}\{\!\{v\}\!\})*\mathsf{credit}(v)\}\ \mathsf{g\_uloado}(v)\ \{\lambda\_.\ \mathsf{obs}(o)\}$$

$$\{\mathsf{lock}(\pi,l,w,i,D,g[v{:=}z])*\mathsf{credit}(v)\}\ \mathsf{g\_dec}(v)\ \{\lambda\_.\ \mathsf{lock}(\pi,l,w,i,D,g[v{:=}z{-}1])\}$$

$$\begin{array}{c}\{\mathsf{lock}(\pi,l,w,i,D,g[v{:=}z])\wedge z{\neq}0\}\ \mathsf{g\_inc}(v)\\ \{\lambda\_.\ \mathsf{lock}(\pi,l,w,i,D,g[v{:=}z{+}1])*\mathsf{credit}(v)\}\end{array}$$

$$\begin{array}{c}\{\mathsf{ulock}(l,w,D)*i(g)\wedge\mathsf{nocrob}(i(g))\wedge\forall v{\in}^1 D.\ g(v){=}|g(v)|\}\ \mathsf{g\_initlock}(l)\\ \{\lambda\_.\ \mathsf{lock}(1,l,w,i,D)\}\end{array}$$

2.7 Verifying the producer/consumer program

The following annotated program indicates how the producer/consumer example can be verified by the presented proof rules[2]. Note that $l{\mapsto}\_$ is a shorthand for $\exists z.\ l\overset{1}{\mapsto}z$, where z is not free in l. The verification of the main routine of the program is started with an empty set of obligations. The waiting levels of the lock and the condition variable are decided to be 1 and 2, respectively. Since the consumer thread needs a credit for $b.v$ to be successfully verified, before forking this thread, the main thread produces a credit $b.v$ by loading $b.v$ to the list of its obligations and supplies the consumer thread with this credit and the half its lock assertion. Then it delegates its $b.v$ obligation to the producer routine, where this obligation is discharged by spending the resulting credit of the producer. At the end of the verification the main routine has no obligation left and consequently is successfully verified. Note that a lock assertion can be divided into half and vice versa, and any assertion can be temporarily removed from the current assertions with the help of the consequence and frame rules.

[2]The current proof rules do not cover the join, loops and the conditional commands. Additionally, they are suitable for the lambda calculus programs. However, the given example conceptually explains how they can be used to verify a specific program.

```
inv(buffer b)(cvar⇀int g) ::= queue(b.q, g(b.v))
routine main()
{
  obs{}
  q := newqueue;
  queue(q, 0) * obs{}
  l := newlock;
  ulock(l, 1, {}) * queue(q, 0) * obs{}
  v := newcvar;
  ulock(l, 1, {(v, 2)}) * queue(q, 0) * obs{}
  b := buffer{q:=q, l:=l, v:=v};
  ulock(b.l, 1, {(b.v, 2)}) * queue(b.q, 0) * obs{}
  g_initlock(b.l);
  lock(1, b.l, 1, inv(b), {(b.v, 2)}) * obs{} where g=0[b.v:=0]
  g_loado(b.v);
  lock(1, b.l, 1, inv(b), {(b.v, 2)}) * obs{b.v} * credit(b.v)
  t := fork (consumer(b));
  lock(0.5, b.l, 1, inv(b), {(b.v, 2)}) * obs{b.v}
  producer(b);
  lock(0.5, b.l, 1, inv(b), {(b.v, 2)}) * obs{}
  join(t);
  lock(1, b.l, 1, inv(b), {(b.v, 2)}) * obs{}
  freecvar(b.v);
  lock(1, b.l, 1, inv(b), {}) * obs{}
  freelock(b.l);
  ∃s. queue(b.q, s) * obs{}
  freequeue(b.q)
  obs{}
}
```

routine producer(buffer b)
{
  lock$(0.5, b.l, 1, \mathsf{inv}(b), \{(b.v, 2)\}) * \mathsf{obs}\{b.v\}$
  lock_acquire$(b.l)$;
  $\exists s.\ \mathsf{locked}(0.5, b.l, 1, \mathsf{inv}(b), \{(b.v, 2)\}, 0[b.v{:=}s]) * \mathsf{queue}(b.q, s) * \mathsf{obs}\{b.v, b.l\}$
  if (sizeof$(b.q) = 0$)
    signalAll$(b.v)$;
    $\exists s.\ \mathsf{locked}(0.5, b.l, 1, \mathsf{inv}(b), \{(b.v, 2)\}, 0[b.v{:=}s{+}1]) * \mathsf{queue}(b.q, s)$
    $*\mathsf{obs}\{b.v, b.l\} * \mathsf{credit}(b.v)$
  if$(b.count \neq 0)$
    g_inc$(b.v)$
    $\exists s.\ \mathsf{locked}(0.5, b.l, 1, \mathsf{inv}(b), \{(b.v, 2)\}, 0[b.v{:=}s{+}1]) * \mathsf{queue}(b.q, s)$
    $*\mathsf{obs}\{b.v, b.l\} * \mathsf{credit}(b.v)$
  enqueue$(b.q, "data")$;
  $\exists s.\ \mathsf{locked}(0.5, b.l, 1, \mathsf{inv}(b), \{(b.v, 2)\}, 0[b.v{:=}s{+}1]) * \mathsf{queue}(b.q, s{+}1)$
  $*\mathsf{obs}\{b.v, b.l\} * \mathsf{credit}(b.v)$
  lock_release$(b.l)$;
  $\mathsf{lock}(0.5, b.l, 1, \mathsf{inv}(b), \{(b.v, 2)\}) * \mathsf{obs}\{b.v\} * \mathsf{credit}(b.v)$
  g_uloado$(b.v)$
  $\mathsf{lock}(0.5, b.l, 1, \mathsf{inv}(b), \{(b.v, 2)\}) * \mathsf{obs}\{\}$
}

routine consumer(buffer b)
{
  $\mathsf{lock}(0.5, b.l, 1, \mathsf{inv}(b), \{(b.v, 2)\}) * \mathsf{credit}(b.v) * \mathsf{obs}\{\}$
  lock_acquire$(b.l)$;
  $\exists s.\ \mathsf{locked}(0.5, b.l, 1, \mathsf{inv}(b), \{(b.v, 2)\}, 0[b.v{:=}s]) * \mathsf{queue}(b.q, s)$
  $*\mathsf{credit}(b.v) * \mathsf{obs}\{b.l\}$
  while(sizeof$(b.q) = 0$)
    $\exists s.\ \mathsf{locked}(0.5, b.l, 1, \mathsf{inv}(b), \{(b.v, 2)\}, 0[b.v{:=}s]) * \mathsf{queue}(b.q, s)$
    $*\mathsf{credit}(b.v) * \mathsf{obs}\{b.l\} \wedge s{=}0$
    wait$(b.v, b.l)$
    $\exists s.\ \mathsf{locked}(0.5, b.l, 1, \mathsf{inv}(b), \{(b.v, 2)\}, 0[b.v{:=}s]) * \mathsf{queue}(b.q, s)$
    $*\mathsf{credit}(b.v) * \mathsf{obs}\{b.l\}$
  $\exists s.\ \mathsf{locked}(0.5, b.l, 1, \mathsf{inv}(b), \{(b.v, 2)\}, 0[b.v{:=}s]) * \mathsf{queue}(b.q, s)$
  $*\mathsf{credit}(b.v) * \mathsf{obs}\{b.l\} \wedge s{\neq}0$
  dequeue$(b.q)$;
  $\exists s.\ \mathsf{locked}(0.5, b.l, 1, \mathsf{inv}(b), \{(b.v, 2)\}, 0[b.v{:=}s]) * \mathsf{queue}(b.q, s{-}1)$
  $*\mathsf{credit}(b.v) * \mathsf{obs}\{b.l\} \wedge s{\neq}0$
  g_dec$(b.v)$
  $\exists s.\ \mathsf{locked}(0.5, b.l, 1, \mathsf{inv}(b), \{(b.v, 2)\}, 0[b.v{:=}s{-}1]) * \mathsf{queue}(b.q, s{-}1) * \mathsf{obs}\{b.l\}$
  lock_release$(b.l)$;
  $\mathsf{lock}(0.5, b.l, 1, \mathsf{inv}(b), \{(b.v, 2)\}) * \mathsf{obs}\{\}$
}

3 Soundness Proof

In this section a high-level soundness proof of the presented approach is provided.

3.1 Safety of Configuration

Definition 1 (Safety of configuration)**.** A configuration is considered to be safe if 1) it does not abort, and 2) it is not a deadlock configuration, and 3) execution of any arbitrary thread leads to a safe configuration too. This property can be formally defined as follows, where for any configuration κ, $\mathsf{safe}_0\ \kappa$ always holds.

$$\mathsf{safe\_conf}\ \kappa \Leftrightarrow \forall n.\ \mathsf{safe}_n\ \kappa$$
$$\mathsf{safe}_n\ \kappa \Leftrightarrow \neg\ \mathsf{abort}\ \kappa\ \wedge\ \mathsf{nodeadlock}\ \kappa\ \wedge\ \forall id\ \kappa'.\ \kappa \rightsquigarrow^{id} \kappa' \Rightarrow \mathsf{safe}_{n-1}\ \kappa'$$
$$\mathsf{nodeadlock}(t,h) \Leftrightarrow t \neq \mathbf{0} \Rightarrow \exists id\ \kappa'.\ (t,h) \rightsquigarrow^{id} \kappa'$$

A configuration is an abort configuration if 1) any thread in this configuration accesses a dangling pointer, or 2) two different threads in the configuration concurrently access and write at the same location, or 3) a thread releases a lock that has not been already acquired, or 4) a thread frees a lock that has been acquired.

$$\mathsf{abort}(t[id{:=}\mathsf{free}(e)], h[[\![e]\!]{:=}\varnothing])$$
$$\mathsf{abort}(t[id{:=}\mathsf{lookup}(e)], h[[\![e]\!]{:=}\varnothing])$$
$$\mathsf{abort}(t[id{:=}\mathsf{mutate}(e_1,e_2)], h[[\![e_1]\!]{:=}\varnothing])$$
$$\mathsf{abort}(t[id{:=}\mathsf{let}\ x{:=}c_1\ \mathsf{in}\ c_2], h) \quad \text{if } \mathsf{abort}(t[id{:=}c_1], h)$$
$$\mathsf{abort}(t[id_1{:=}c_1, id_2{:=}c_2], h) \quad \text{if } \mathsf{access}(c_1){\neq}\varnothing \wedge \mathsf{access}(c_1){=}\mathsf{write}(c_2) \wedge id_1{\neq}id_2$$
$$\mathsf{abort}(t[id{:=}\mathsf{acquire}(a)], h[a{:=}\varnothing])$$
$$\mathsf{abort}(t[id{:=}\mathsf{release}(a)], h[a{\neq}0])$$
$$\mathsf{abort}(t[id{:=}\mathsf{freelock}(a)], h[a{\neq}1])$$

3.2 Soundness of the Proof Rules

For simplicity's sake in the rest of this section we only prove the deadlock freedom of programs and focus on the most important parts of the proof. However, a comprehensive formal proof written in Coq[3] can be found on `https://people.cs.kuleuven.be/~jafar.hamin/dlfree/`[4].

Theorem 2 (Soundness of the proof rules)**.**

$$\{\mathsf{obs}\{\}\}\ c\ \{\lambda\_.\ \mathsf{obs}\{\}\} \Rightarrow \mathsf{safe\_conf}\ (\mathbf{0}[id{:=}c], \mathbf{0})$$

Proof. The soundness theorem is proved by introducing two intermediate safety definition, namely safec and safet. The former deals with a single thread and its local permission heap, and the latter deals with a set of threads and their local permission heap. Lemmas 3, 5, and 7 are some high-level auxiliary lemmas used to prove this theorem. □

[3]Some parts of this formalization are inspired by [5]
[4]This formalization is still under development

Lemma 3 (Soundness of the proof rules w.r.t. the safety of commands)**.**

$$\{P\}\ c\ \{Q\} \Rightarrow m, d, p \Vdash P \Rightarrow \mathsf{safec}_n\ c\ p\ m\ d\ Q$$

Proof. By case analysis of the proof rules and induction on n, if required. □

Definition 4 (Safety of Commands)**.** $\mathsf{safec}_n\ c\ p\ m\ d\ a$. A command c in a local permission heap p with local obligations m and local credits d and postcondtion a is considered to be safe if and only if:

- Either $n = 0$
- Or $c = \mathsf{val}(z)$ and $m, d, p \Vdash a(z)$
- Or c is neither a value nor a ghost command and $n{>}0$ and also 1) if c is trying to acquire a lock z or waiting for a condition variable z then the waiting level of z is lower than the waiting levels of all obligations in m; and 2) if c is waiting for a condition variable v then there exists at least one credit for v in d; and 3) the following invariants hold during the execution of c: 3-1) for any lock l that has been acquired, l is in the (multi)set of global obligations, and 3-2) for any condition vairable v for which a thread is waiting, v is in the set of global obligations. The second invariant follows from the constraint number 2 and the following auxiliary invariants: 3-2-1) $\forall v.\ D(v) \leqslant O(v) + cvg(v)$, where $D(v)$ and $O(v)$ are the number of occurrences of the condition variable v in the set of global credits and global obligations, respectively, and $cvg(v)$ is the cvg value of v, and 3-2-2) for any condition variable v for which a thread is waiting, $cvg(v) \leqslant 0$.
- Or c is a ghost command and $\mathsf{safe}_{n-1}\ c'\ p'\ m'\ d'\ a$ holds, where c', p', m' and d' are the effect of execution of the ghost command in c, p, m, and d, respectively.

Lemma 5 (Soundness of the safety of commands w.r.t. the safety of threads)**.**

$$\begin{array}{l}\mathsf{safec}_n\ c\ p_c\ o_c\ d_c\ Q\ \wedge\ Q \models \mathsf{obs}\{\}\ \wedge\ \mathsf{safet}_n\ t[id{:=}\varnothing]\ p_t\ o_t\ d_t \\ \quad \Rightarrow \mathsf{safet}_n\ t[id{:=}c]\ p_t \oplus p_c\ o_t \uplus \lceil o_c \rceil^{id}\ d_t \uplus d_c\end{array}$$

where $\lceil o_c \rceil^{id}$ *changes each element* $z \in o_c$ *to* (id, z).

Proof. By induction on n. □

Definition 6 (Safety of threads)**.** $\mathsf{safet}_n\ t\ p\ o\ d$. The set of threads t in a permission heap p with o as a set of thread id-obligation pairs, and d as a set of credits is considered to be safe if and only if:

- Either $n = 0$
- Or $t = \mathsf{0}$ and o is empty

- Or $n{>}0$ and 1) if any thread *tid* is trying to acquire a lock z or waiting for a condition variable z then the waiting level of z is lower than the waiting levels of obligations of *tid* in o; and 2) for any thread waiting for a condition variable v there exists at least one credit for v in d; and 3) the invariants mentioned in the definition of safety of commands also hold during the execution of any thread in t.

Lemma 7 (Soundness of the safety of threads)**.**

$$\mathsf{safet}_n\ t\ p\ o\ d \Rightarrow \mathsf{safe}_n\ t\ \lceil p \uplus p_{inv} \rceil$$

where p_{inv} provides the invariants of all the locks in p and p_{inv} that have not been acquired, and $\lceil p \rceil$ converts the permission heap p to its coresponding heap.

Proof. By induction on n and lemma 8. □

Lemma 8 (Valid configuration does not deadlock)**.** *Assuming a given configuration extended by a set of thread-obligation pairs, namely o, such that 1) for any lock z that has been acquired or any condition vairable z for which a thread is waiting, there exists a thread tid such that $(tid, z){\in}o$; and 2) for any $(tid, z){\in}o$ the thread tid is not terminated; and 3) if any thread tid in this configuration is trying to acquire a lock z or waiting for a condition variable z then the waiting level of z is lower than the waiting levels of obligations of tid in o; then this configuration does not deadlock.*

Proof. By contradiction: We assume that there is no progress in the configuration but there is still a thread *id* such that $t(id){=}c$. Assuming that heap and thread pool have an infinite capacity, c is either $c{=}\mathsf{acquire}(z)$, where z has been already acquired, or $c{=}\mathsf{sleep}(z, l)$. By 1, o contains at least one element, and consequently, there should exist some $(tid_{min}, z_{min}){\in}o$ such that the waiting level of z_{min}, namely w_{min}, is the minimum waiting level of all obligations in o. By 2, there should exist a command c' such that $t(tid_{min}){=}c'$, and since there is no progress in this configuration there should exist a resource z with waiting level w_z such that c' is either $\mathsf{acquire}(z)$, where z has been already acquired, or $\mathsf{asleep}(z, l')$. By 1, there is an obligation for z and by 3, w_z should be strictly lower than the waiting levels of all obligations of tid_{min}, including z_{min}. Consequently, $w_z{<}w_{min}$ that contradicts the assumption claiming w_{min} is the minimum waiting level of all obligations. □

References

[1] Bart Jacobs, Dragan Bosnacki, and Ruurd Kuiper. Modular termination verification. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[2] Pedro da Rocha Pinto, Thomas Dinsdale-Young, Philippa Gardner, and Julian Sutherland. Modular termination verification for non-blocking concurrency. In *European Symposium on Programming Languages and Systems*, pages 176–201. Springer, 2016.

[3] Bart Jacobs. Provably live exception handling. In *Proceedings of the 17th Workshop on Formal Techniques for Java-like Programs*, page 7. ACM, 2015.

[4] Pontus Boström and Peter Müller. *Modular verification of finite blocking in non-terminating programs*, volume 37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[5] Viktor Vafeiadis. Concurrent separation logic and operational semantics. *Electronic Notes in Theoretical Computer Science*, 276:335–351, 2011.